C/C++ Contributing Editors

The New C: Integers, Part 3

Randy Meyers

At first glance, C99's new integral types seem to threaten its portability. But a few added headers and typedefs improve the outlook dramatically.

In my January 2001 column, I discussed how the introduction of 64-bit machines motivated C99 to generalize the rules for integer types and to allow new integer types as extensions. But, C99 did not describe any syntax for naming these new integer types. This seems to put programmers in an awkward position. Your implementation might support additional integer types. You might find such types useful. However, lacking a standardized syntax, it appears that you must learn about implementation-specific extensions to use new integer types, and that such uses automatically eliminate your chances for portability. This month, we see that C99 actually provides a solution to this problem that might even help with related portability problems. Interestingly, the 64-bit machines again thrust the issue before the standards committee.

Integers and Portability

Initially, the vendors of 64-bit hardware and software disagreed wildly about the mapping from C keywords to integer types. While much of the focus was on the sizes of the **int** and **long** types, some vendors even wanted to change the size of **short**. By contemplating changing the mapping to integer types, the 64-bit vendors magnified an old problem in writing portable C. Which keyword should you use when you want an integer of a particular size?

The C89 and C99 Standards do make certain guarantees. For example, **short** and **int** are at least a 16-bit integer type, and **long** is at least a 32-bit integer type. Thus, if you specifically want a 32-bit type, you might be tempted to just use **long**. However, on most 64-bit machines, **long** is 64 bits. If this causes problems for you, there is a simple solution. Use a typedef for your type, and change the definition of the typedef when you port your program.

The C committee realized that this simple technique can produce high payoff for little work, and that there was an advantage in standardizing the names of the typedefs. Thus, two new standard headers were added to C99, **<stdint.h>** and **<inttypes.h>**. Some C and C++ implementations have been providing these headers for years. If your implementation does not provide them, and they would be useful to

you, they are simple enough that you could write your own version of the headers. They are not in ANSI C++98, but they are likely to be part of a future revision.

Header <stdint.h>

The header **<stdint.h>** defines typedefs for integers of various sizes, macros that expand into the maximum value (and for the signed types, the minimum value) for those types, and a few other utility macros.

Most of the typedefs in the header are named according to the following patterns:

```
int_leastN_t
uint_leastN_t
int_fastN_t
uint_fastN_t
intN_t
uintN_t
```

where *N* is replaced with a decimal integer giving the size in bits (excluding pad bits) of that integer type. Typically, you will find typedefs defined in the header where *N* in the above patterns has been replaced by 8, 16, 32, and 64, yielding 24 different typedef names. The Standard permits an implementation to define other typedefs in the header fitting the patterns. The typedefs may be defined to be the traditional integer types or any extended integer types supported by the implementation. For example, an implementation supporting 128-bit integers would provide int128_t, uint128_t, and so on.

The above typedef names that start with "u" are typedefs for unsigned types. The typedef names that do not start with "u" are the corresponding signed types. The typedef names break down into three families.

The first family is the types formed from the patterns that contain "least" in their names. These types are at least the specified size in bits given by *N*, but they might be larger if the hardware makes it necessary. For example, on a 36-bit machine where int was 36 bits and short was 18 bits, the header would declare:

```
typedef short int_least16_t;
typedef int int_least32_t;
typedef short int_least18_t;
typedef int int_least36_t;
```

As we will see below, the first two of these typedefs are required. The last two of these typedefs are the implementation taking advantage of the ability to add extra typedefs to the header for "extra" integer types it supports. The least types are required to be the smallest integer type holding the required number of bits. Thus, the least types are the most space efficient types that can represent integers with the requested number of bits.

The second family is the types formed from the patterns that contain "fast" in their names. Like the least types, these types can represent an integer with the requested number of bits, but the fast types may be as large as necessary to provide efficient computation. Consider a 64-bit machine that can operate on 16-bit integers, but whose operations on 16-bit integers are more expensive than operations on a 32- or 64-bit integer. Assume that short is 16 bits and int is 32-bits on that machine. The header would declare:

```
typedef short int_least16_t;
typedef int int_fast16_t;
```

Both the least and fast types can store all of the bits of a 16-bit integer, but the least types are optimized for space (like a big array of integers) while the fast types are optimized for speed (like a loop index).

The third family is the types that contain neither "least" or "fast" in their names. These are the exact-sized, two's complement integers with no padding bits. For the least and fast types, the Standard requires implementations to provide typedefs where *N* has been replaced with 8, 16, 32, and 64. However, for the exact-sized types, the requirements on the types are so demanding that the Standard makes these types optional. (This is discussed further below.) These types should be used sparingly. Typical uses are laying out a struct to match an externally defined data layout, such as a binary file from another system or a network packet. A 36-bit machine would not be able to provide typedefs where *N* was 8, 16, 36, and 64, but would be able to provide names where *N* was 9, 18, 36, and 72.

An implementation is required to define macros in <stdint.h> giving the maximum (and for signed types, the minimum) values that can be stored in the types defined in the header. The name patterns for these limit macros match the patterns for the typedef names:

```
INT_LEASTN_MAX
INT_LEASTN_MIN
UINT_LEASTN_MAX
INT_FASTN_MAX
INT_FASTN_MIN
UINT_FASTN_MAX
INTN_MAX
INTN_MIN
UINTN_MAX
```

Note that an implementation defines limit macros if and only if the corresponding typedef is defined. Thus, you can use the limit macros to test whether a typedef name is defined. For example:

```
#include <stdint.h>
#ifdef INT_LEAST24_MAX
int_least24_t x;
#else
int_least32_t x;
#endif
```

The above defines x to have type int_least24_t if that type is supported. Otherwise, x is defined to have the type int_least32_t, a type the Standard requires to exist.

The <stdint.h> header also defines two function-like macros for forming integer constants whose return type is one of the int_least*N*_t or uint_least*N*_t types, respectively:

INT*N*_C(constant) UINT*N*_C(constant)

where *N* is replaced with a decimal integer corresponding to one of the "least" types. The argument to these macros should be an unsuffixed constant. The macro uses the preprocessor paste operator ## to add a constant suffix to produce a constant with the proper type. For example, UINT64_C(0xA) might expand to 0xAULL, if the uint_least64_t type was unsigned long long.

Special Integer Types

The <stdint.h> header also defines a few integer types useful for special purposes.

The typedefs intptr_t and uintptr_t are respectively a signed integer type and the corresponding unsigned integer type large enough to hold a pointer to an object without losing any information. (These integer types might not be large enough to hold a pointer to a function.) Specifically, the Standard demands that if you cast a pointer to void to one of these integer types, then cast the integer back to pointer to void, that the result of the two casts should equal the original pointer. There are a few systems where no such integer types exist, so C99 makes these typedefs optional in the header. You can test whether these types are declared by testing if their limit macros are defined: INTPTR_MIN, INTPTR_MAX, and UINTPTR_MAX.

Perhaps the two most important typedefs defined in <stdint.h> are intmax_t and uintmax_t. These are respectively the largest signed integer type and the corresponding unsigned integer type supported by the implementation. Obviously such types are useful whenever you want your program to be able to process the largest numbers supported on the machine. As a less obvious use, such types are handy when working with typedefs for integer types from different sources. Consider:

```
apple_t num_apples();
orange_t num_oranges();
intmax_t num_fruit;
num_fruit = num_apples() + num_oranges();
```

where apple_t is a signed integer typedef used for storing the number of apples, and orange_t is a signed integer typedef used for storing the number of oranges. If you want to store the total number of fruit, neither apple_t nor orange_t might be appropriate. Perhaps you do not have many apples so that apple_t is short while you do have lots of oranges so that orange_t is long. Perhaps the situation is reversed. In the absence of a typedef specifically for storing numbers of any type of fruit, using the largest integer type, intmax_t, is a good idea. Luckily, comparing apples to oranges causes no similar problems.

There are limit macros for intmax_t and uintmax_t: INTMAX_MIN, INTMAX_MAX, and UINTMAX_MAX. There are also function-like macros, similar to the ones described above, for writing constants of type intmax_t and uintmax_t: INTMAX_C(constant) and UINTMAX_C(constant). The Standard requires that intmax_t, uintmax_t, and their limit and constant macros to be defined. On many implementations, intmax_t will be long long and uintmax_t will be unsigned long long .

Finishing out <stdint.h> are limit macros for integer typedefs defined in other headers. The other headers failed to define macros giving the minimum and maximum values. Rather than add new names to popular headers from C89, C99 defined the limit macros in <stdint.h>.

For ptrdiff_t, the limits are PTRDIFF_MIN and PTRDIFF_MAX. For sig_atomic_t, the limits are SIG_ATOMIC_MIN and SIG_ATOMIC_MAX. For size_t, the limit is SIZE_MAX. For wchar_t, the limits are WCHAR_MIN and WCHAR_MAX. For wint_t, the limits are WINT_MIN and WINT_MAX.

Header <inttypes.h>

The header <inttypes.h> includes the header <stdint.h>, and then defines a few functions (which I will cover in a future column) and many additional macros. Even though <inttypes.h> includes <stdint.h>, you may explicitly include both headers if you wish. All standard C99 headers except <assert.h> may be included multiple times without problem.

The main purpose of <inttypes.h> is to allow printf and scanf to be used with the integer types defined in <stdint.h>. This is accomplished by defining macros that expand into quoted strings that are a particular printf or scanf format conversion specifier prefixed with any needed length modifiers. I will discuss the patterns of these macro names for format conversion specifiers later. For now, be aware that there is a separate macro for every integer type in <stdint.h> and

for every printf and scanf format conversion specifier that operates on integers, and that there is a separate such macro for printf versus scanf.

Using string literal concatenation, these macros can be used to form a printf or scanf format string. String literal concatenation is the C feature where two string literals separated only by whitespace will be combined into one large string literal by the compiler. Consider the small program in Listing 1. <inttypes.h> defines PRIdFAST64, which expands to the printf d conversion specifier for printing an intfast64_t, and **SCNdFAST64**, which expands to the **scanf d** conversion specifier for reading an **intfast64_t**. String literal concatenation combines the string literals from the two macros with the other adjacent string literals to produce a single string literal for the **scanf** format and a string literal for the **printf** format. Assuming the **int_fast64_t** type is really **long long**, both of the two macros expand into the quoted string **"lld"**, which is the **d** format conversion specifier prefixed by the **ll** modifier to say the type is **long long**. Thus, the resulting format string for **scanf** is **"%lld"** and the resulting format string for **printf** is **"x=%08lld\n"**.

Note that a leading percent sign is not part of the strings from the conversion specifier macros. Thus, you can specify special flags, width, or precision arguments to the conversion specifier in the string concatenated to the front of the string from the macro. In the **printf** format for Listing 1, the leading zero flag was given to cause leading zeros to be written in front of the number, and a field width of 8 was specified.

The macros for **printf** format conversion specifiers follow the naming patterns:

PRIs*N*
PRIsLEAST*N*
PRIsFAST*N*
PRIsMAX
PRIsPTR

where *s* is replaced by a printf integer format conversion specifier, one of d, i, o, u, x, or X, and *N* is replaced by a decimal number that is the same *N* used to form typedef names in <stdint.h>. The first macro name pattern above is used with the exact width types. The pattern containing "LEAST" is used with the least integer types. The pattern containing "FAST" is used with the fast integer types. The pattern containing "MAX" is used with the intmax_t and uintmax_t types. The pattern containing "PTR" is used with the intptr_t and uintptr_t types.

The macros for the scanf format conversion specifiers follow the same naming patterns as printf, except that the macros begin with SCN instead of PRI, and *s* cannot be replaced with X. There are separate macros for printf and scanf because printf versus scanf format conversion specifiers sometimes need different length modifiers for the same type. For example, you print a short with %d and read it with %hd.

Just as intmax_t and uintmax_t are useful when storing integers of unknown size, they can be useful when printing an integer of unknown size. Assume that

the variable x has some signed integer type, but you do not know which type. You can print x by casting it to intmax_t before printing it:

```
printf("%" PRIdMAX, (intmax_t) x);
```

The <inttypes.h> header was invented by some of the 64-bit vendors and predates C99. Originally, <inttypes.h> directly included the contents of <stdint.h>. The C99 committee decided to divide the header into two in order to separate the printf and scanf format macros (which do not interest C++ programmers) from the integer types and limit macros (which do interest C++ programmers). Some systems that are not fully compatible with C99 lack a <stdint.h> but have the older version of <inttypes.h>.

Conclusion

As many programmers have independently discovered, a simple solution to portability and different integer types is to use typedefs. C99 has standardized the names and uses of such typedefs in the new headers <stdint.h> and <inttypes.h>. These headers not only solve the problem of the mapping from C keywords to different sized integers, but also might give you access to an implementation's extended integer types in a way that does not automatically disallow portability. These headers are also simple enough that if your implementation does not yet provide them, it is a simple matter to write your own.

Randy Meyers is consultant providing training and mentoring in C, C++, and Java. He is the current chair of J11, the ANSI C committee, and previously was a member of J16 (ANSI C++) and the ISO Java Study Group. He worked on compilers for Digital Equipment Corporation for 16 years and was Project Architect for DEC C and C++. He can be reached at rmeyers@ix.netcom.com.